TEXAS A&M UNIVERSITY

Inter Process Communication: Pipes and FIFO

CSCE 313: Introduction to Computer Systems

Outline

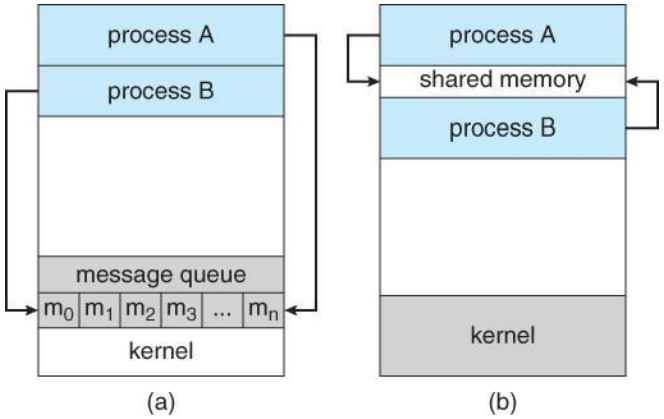
1. Pipe system calls
2. Fork system calls
3. FIFO (named pipes)

Real-time communication between programs?

Inter-process communication (IPC) is a mechanism that allows processes to communicate with each other and synchronize their actions.

Processes can communicate with each other through both:

1. Shared Memory
2. Message passing



In this module we will focus on message passing mechanisms.

What is a pipe in Linux?

A pipe is a construct in the Unix OS that allows two processes to communicate.

One end of the pipe is the read end; the other is the write end.

A pipe is created via a pipe() system call.

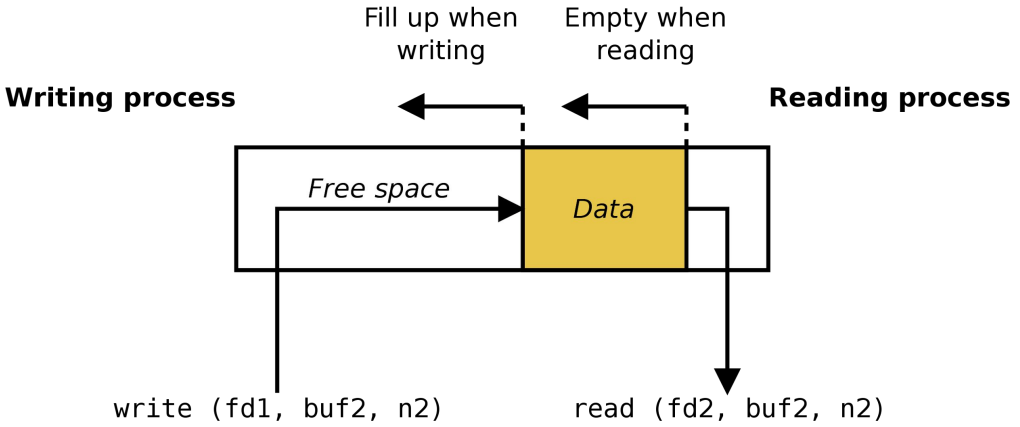
```
int pipe(int fd[2])
```

The function returns a pair of file descriptors*

- fd[0] is the read end
- fd[1] is the write end

Data is received in the order it is sent

$ man 2 pipe



How to create a pipe?

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char* argv[]){
   int fd[2];
   char buf[30];
   //create pipe
   if(pipe(fd) == -1){
      perror("pipe");
      exit(EXIT_FAILURE);
   }
   //write to pipe
   printf("writing to file descriptor #%d\n", fd[1]);
   write(fd[1],"CSCE 313",9);
   //read from pipe
   printf("reading from file descriptor #%d\n", fd[0]);
   read(fd[0], buf,9);
   printf("read \"%s\"\n",buf);
   return 0;
}
```

pipe() takes an array of two ints as an argument.

What are the return values of pipe()?

-1: pipe failure

0: pipe success

pipe() fills in the array with two file descriptors.

How to use a pipe?

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char* argv[]){
   int fd[2];
   char buf[30];
   //create pipe
   if(pipe(fd) == -1){
      perror("pipe");
      exit(EXIT_FAILURE);
   }
   //write to pipe
   printf("writing to file descriptor #%d\n", fd[1]);
   write(fd[1],"CSCE 313",9);
   //read from pipe
   printf("reading from file descriptor #%d\n", fd[0]);
   read(fd[0], buf,9);
   printf("read \"%s\"\n",buf);
   return 0;
}
```

When any bytes are written to fd[1],the OS makes them available for reading from fd[0].

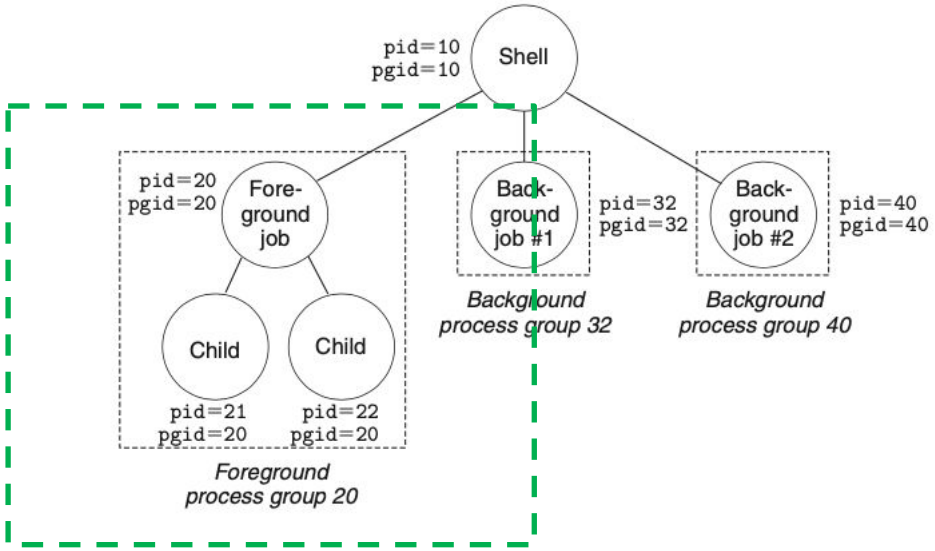
Linux pipe command

Sending Signals from the Keyboard

Unix shells use the abstraction of a job to represent the processes that are created as a result of evaluating a single command line. For example, typing

linux> ls | sort

This command creates a foreground job consisting of two processes connected by a Unix pipe: one running the ls program, the other running the sort program.



Linux pipe command example 1 (`cmd1 | cmd2`)

```
davidkebo@linux:~$ cat games.txt
Horizon Forbidden West
Spider-Man: Miles Morales
Deathloop
Ratchet and Clank: Rift Apart
Elden Ring
Returnal
Sackboy: A Big Adventure
Ghost of Tsushima: Director's Cut
Assassin's Creed Valhalla
Hitman 3
Marvel's Guardians of the Galaxy
Stray
Astro's Playroom
Kena: Bridge of Spirits
Demon's Souls
```

```
davidkebo@linux:~$ cat games.txt | less
Horizon Forbidden West
Spider-Man: Miles Morales
Deathloop
Ratchet and Clank: Rift Apart
Elden Ring
Returnal
Sackboy: A Big Adventure
Ghost of Tsushima: Director's Cut
Assassin's Creed Valhalla
Hitman 3
Marvel's Guardians of the Galaxy
Stray
Astro's Playroom
Kena: Bridge of Spirits
Demon's Souls
(END)
```

Linux pipe command example 2 (`cmd1 | cmd2`)

```
davidkebo@linux:~$ cat games.txt
Horizon Forbidden West
Spider-Man: Miles Morales
Deathloop
Ratchet and Clank: Rift Apart
Elden Ring
Returnal
Sackboy: A Big Adventure
Ghost of Tsushima: Director's Cut
Assassin's Creed Valhalla
Hitman 3
Marvel's Guardians of the Galaxy
Stray
Astro's Playroom
Kena: Bridge of Spirits
Demon's Souls
```

```
davidkebo@linux:~$ cat games.txt | grep Hitman
Hitman 3
```

Linux pipe command example 3 (`cmd1 | cmd2`)

```
davidkebo@linux:~$ cat games.txt
Horizon Forbidden West
Spider-Man: Miles Morales
Deathloop
Ratchet and Clank: Rift Apart
Elden Ring
Returnal
Sackboy: A Big Adventure
Ghost of Tsushima: Director's Cut
Assassin's Creed Valhalla
Hitman 3
Marvel's Guardians of the Galaxy
Stray
Astro's Playroom
Kena: Bridge of Spirits
Demon's Souls
```

```
davidkebo@linux:~$ cat games.txt | sort
Assassin's Creed Valhalla
Astro's Playroom
Deathloop
Demon's Souls
Elden Ring
Ghost of Tsushima: Director's Cut
Hitman 3
Horizon Forbidden West
Kena: Bridge of Spirits
Marvel's Guardians of the Galaxy
Ratchet and Clank: Rift Apart
Returnal
Sackboy: A Big Adventure
Spider-Man: Miles Morales
Stray
```

Linux pipe command example 4 (`cmd1 | cmd2`)

```
davidkebo@linux:~$ cat games.txt
Assassin's Creed Valhalla
Astro's Playroom
Deathloop
Demon's Souls
Elden Ring
Ghost of Tsushima: Director's Cut
Hitman 3
Horizon Forbidden West
Kena: Bridge of Spirits
Marvel's Guardians of the Galaxy
Ratchet and Clank: Rift Apart
Returnal
Sackboy: A Big Adventure
Spider-Man: Miles Morales
Stray
Elden Ring
```

```
davidkebo@linux:~$ cat games.txt | sort | uniq | head -3 >
top3.txt
davidkebo@linux:~$ cat top3.txt
Assassin's Creed Valhalla
Astro's Playroom
Deathloop
```

Note: Do not confuse pipe (|) redirection with file redirection (>) and (<)

Linux pipe command example 5 (`cmd1 | cmd2`)

Counting files and directories are listed.

The wc command counts the number words in a files specified by the file arguments.

```
davidkebo@linux:~$ ls
Desktop    Music      Templates  content.txt  src-cloud.zip
Documents  Pictures   Videos     games.txt    top3.txt
Downloads  Public     code       src-cloud    topics.txt
davidkebo@linux:~$ ls | wc -l
15
```

Linux pipe command example 6 (`cmd1 | cmd2`)

```
davidkebo@linux:~$ ls -l
total 728
drwxr-xr-x 2 davidkebo davidkebo   4096 Aug 23 04:24 Desktop
drwxr-xr-x 2 davidkebo davidkebo   4096 Aug 23 04:24 Documents
drwxr-xr-x 2 davidkebo davidkebo   4096 Aug 23 04:24 Downloads
drwxr-xr-x 2 davidkebo davidkebo   4096 Aug 23 04:24 Music
drwxr-xr-x 2 davidkebo davidkebo   4096 Aug 23 04:24 Pictures
drwxr-xr-x 2 davidkebo davidkebo   4096 Aug 23 04:24 Public
drwxr-xr-x 2 davidkebo davidkebo   4096 Aug 23 04:24 Templates
drwxr-xr-x 2 davidkebo davidkebo   4096 Aug 23 04:24 Videos
drwxrwxr-x 7 davidkebo davidkebo   4096 Sep  5 15:24 code
-rw-rw-r-- 1 davidkebo davidkebo    108 Sep  8 03:23 content.txt
-rw-rw-r-- 1 davidkebo davidkebo    308 Sep  8 13:59 games.txt
drwxrwxr-x 3 davidkebo davidkebo   4096 Dec 15  2020 src-cloud
-rw-rw-r-- 1 davidkebo davidkebo 686475 Aug 23 03:14 src-cloud.zip
-rw-rw-r-- 1 davidkebo davidkebo     53 Sep  8 14:08 top3.txt
-rw-rw-r-- 1 davidkebo davidkebo    188 Sep  8 13:33 topics.txt
```

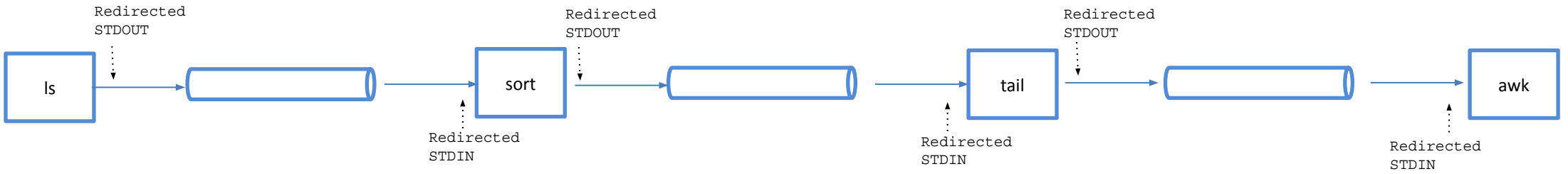
```
davidkebo@linux:~$ ls -l | sort -n -k 5 | tail -n 1 | awk '{print $NF}'
src-cloud.zip
```

1. The command above creates four processes linked together.
2. ls -l prints the list of files with details.
3. The list is sent as input to sort, which sorts numerically based on the 5th field (the file size).
4. The tail process grabs the last line, which is the line for the largest file.
5. awk prints the last field of that line, which is the file name of the largest file.

Linux pipe command example 6 (`cmd1 | cmd2`)

```
davidkebo@linux:~$ ls -l | sort -n -k 5 | tail -n 1 | awk '{print $NF}'
src-cloud.zip
```

1. The command above creates four processes linked together.
2. ls -l prints the list of files with details.
3. The list is sent as input to sort, which sorts based on the 5th field (the file size).
4. The tail process grabs the last line, which is the line for the largest file.
5. awk prints the last field of that line, which is the file name of the largest file.



The figure above is the chained logical structure of the sequence of bash commands connected with pipes.

UNIX fork()

fork() is a system call that creates a new process by duplicating the calling process.

The process calling fork() is the parent process.

The new process is the child process.

```
pid_t fork(void);
```

What are the return values of fork()?

< 0 : fork failure - child process not created

= 0 : fork success - value 0 returned to the child process

> 0 : fork success - process ID of the child process returned to the parent process.

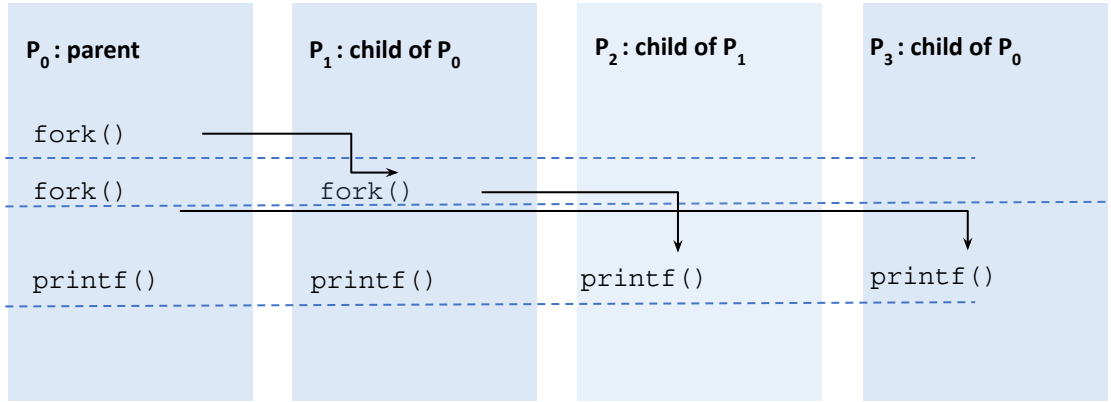
UNIX fork() example 1

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
  fork();

  printf("Welcome to CSCE 313!\n");
  return 0;
}
```

```
davidkebo@CSCE-C02F159MMD6M forks % ./a.out
Welcome to CSCE 313!
Welcome to CSCE 313!
```

UNIX fork() example 2

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
   fork();
   fork();

   printf("Welcome to CSCE 313!\n");
   return 0;
}
```

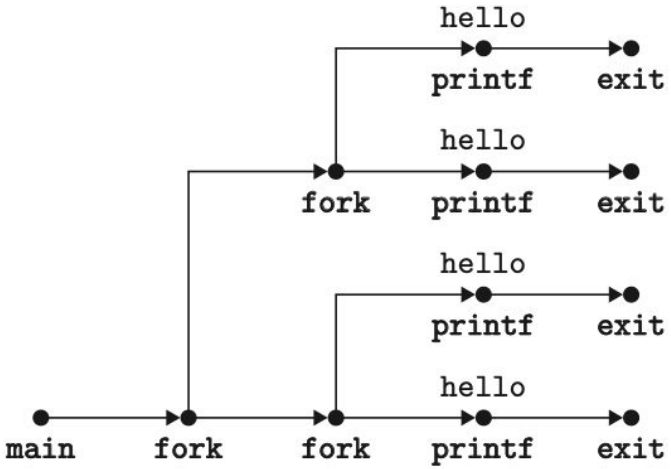



```
davidkebo@CSCE-C02F159MMD6M forks % ./a.out
Welcome to CSCE 313!
Welcome to CSCE 313!
Welcome to CSCE 313!
Welcome to CSCE 313!
```

The number of times the message prints is equal to number of process created.
2^n (n is the number of fork system calls)

UNIX fork() process graph

```
1   int main()
2   {
3       Fork();
4       Fork();
5       printf("hello\n");
6       exit(0);
7   }
```



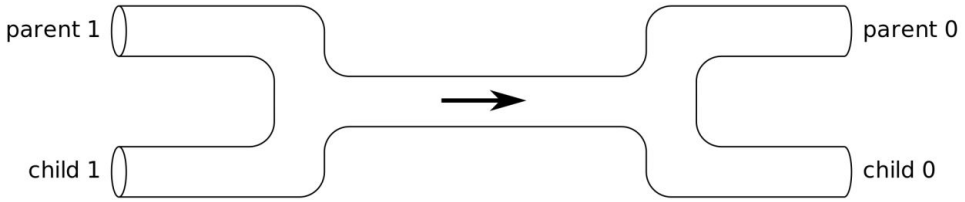
fork() and pipe()

It is not useful for one process to use a pipe to talk to itself. Typically, a process creates a pipe just before it forks more child processes.

The pipe is then used for communication either between the parent or child processes, or between two sibling processes.

In the following program the parent writes a message to the pipe.

The child reads from the pipe 1 byte at a time until the pipe is empty.



fork() and pipe() example 1

```c
/* pipe3.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
int main(int argc, char* argv[])
{
    int pipefds[2];
      pid_t pid;
    char buf[30];
    //create pipe
    if(pipe(pipefds) == -1){
        perror("pipe");
        exit(EXIT_FAILURE);
    }
      memset(buf,0,30);
      pid = fork();
    if (pid > 0) {
      printf(" PARENT write in pipe\n");
    //parent close the read end
          close(pipefds[0]);
    //parent write in the pipe write end
      write(pipefds[1], "CSCI3150", 9);
    //after finishing writing, parent close the write end
          close(pipefds[1]);
    //parent wait for child
      wait(NULL);
    }else {
      //child read from the pipe read end until the pipe is empty
      while(read(pipefds[0], buf, 1)==1)
        printf("CHILD read from pipe -- %s\n", buf);
      //after finishing reading, child close the read end
      close(pipefds[0]);
      printf("CHILD: EXITING!");
      exit(EXIT_SUCCESS);
    }
    return 0;
}
```

```
davidkebo@CSCE-C02F159MMD6M pipes % ./a.out
PARENT: writing to the pipe
CHILD read from pipe --C
CHILD read from pipe --S
CHILD read from pipe --C
CHILD read from pipe --I
CHILD read from pipe --3
CHILD read from pipe --1
CHILD read from pipe --5
CHILD read from pipe --0
CHILD read from pipe --
```

Problem:

The child process doesn't exit because pipefds[1] is open. The system assumes that a write could occur while the write end is still open, and the system will not report EOF.

The child blocks and waits to read from pipefds[0], and the operating system doesn't know that no process will be writing to pipdfds[1].

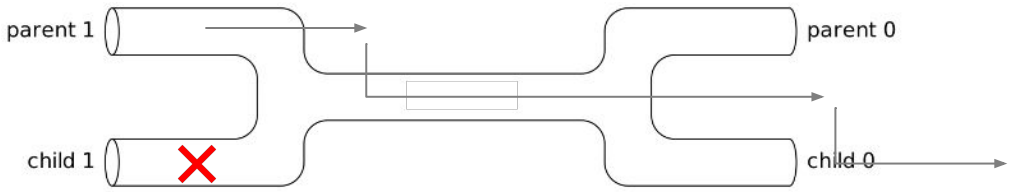
fork() and pipe() example 2

```c
int main(int argc, char* argv[])
{
    int pipefds[2];
      pid_t pid;
    char buf[30];
    //create pipe
    if(pipe(pipefds) == -1){
        perror("pipe");
        exit(EXIT_FAILURE);
    }
      memset(buf,0,30);
      pid = fork();
    if (pid > 0) {
      printf(" PARENT write in pipe\n");
    //parent close the read end
          close(pipefds[0]);
    //parent write in the pipe write end
      write(pipefds[1], "CSCI3150", 9);
    //after finishing writing, parent close the write end
          close(pipefds[1]);
    //parent wait for child
      wait(NULL);
    }else {
      //child close the write end
      close(pipefds[1]);   //-----line *
      //child read from the pipe read end until the pipe is empty
      while(read(pipefds[0], buf, 1)==1)
        printf("CHILD read from pipe -- %s\n", buf);
      //after finishing reading, child close the read end
      close(pipefds[0]);
      printf("CHILD: EXITING!");
      exit(EXIT_SUCCESS);
    }
    return 0;
}
```

```
davidkebo@CSCE-C02F159MMD6M week3 % ./a.out
PARENT: write in pipe.
CHILD read from pipe --C
CHILD read from pipe --S
CHILD read from pipe --C
CHILD read from pipe --I
CHILD read from pipe --3
CHILD read from pipe --1
CHILD read from pipe --5
CHILD read from pipe --0
CHILD read from pipe --
CHILD: EXITING!
```

Note:

When the child process finishes reading all the bytes of data, we close the related file descriptors.

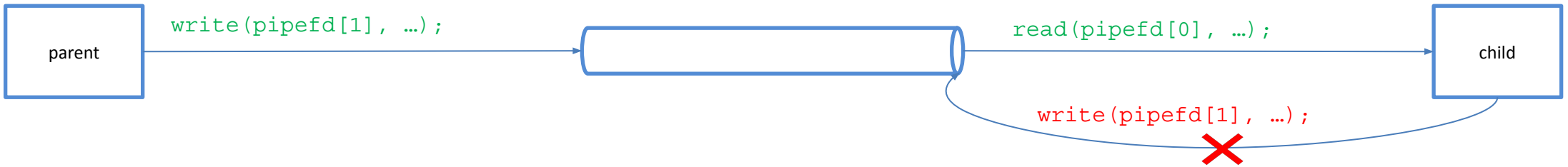


Pipes for bidirectional communication

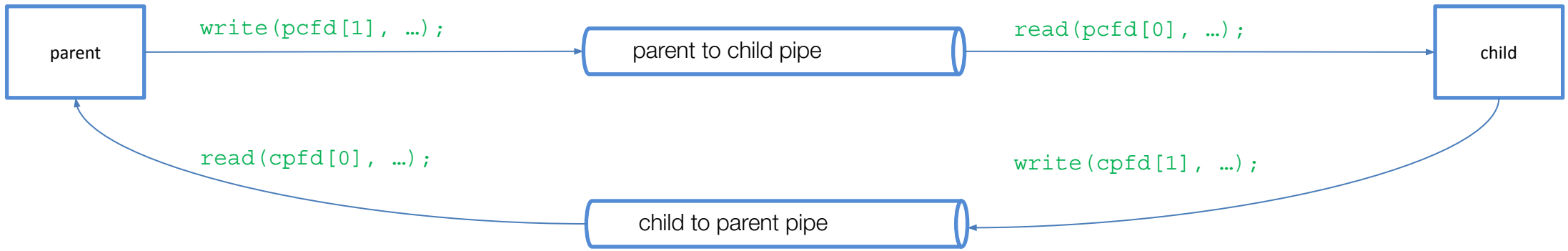
Problem:

Pipes are unidirectional and should not be used to write back.

Using a single pipe for bidirectional communication will fail.



Pipes for bidirectional communication



We can use two pipes for bidirectional communication between parent and child.

1. The parent process uses pcfd to send data to the child process.
2. The child process uses cpfd to send data to the parent process.

In both cases, the calls to write() use index 1 and the read() calls use index 0 on the file descriptor array.

Copy of a file descriptor: dup2()

The dup2() system call creates a copy of a file descriptor.

- If the copy is successfully created, then the original and copy file descriptors may be used interchangeably.
- They both refer to the same open file description and thus share file offset and file status flags.

```
int dup2 (int oldfd, int newfd);
```

oldfd: old file descriptor

newfd: new file descriptor which is used by dup2() to create a copy.

Copy of a file descriptor: dup2()

Bash uses dup2() with pipes to link commands together.

Example: ls | sort

1. The ls process closes its read end of the pipe and links the write end to its standard output.
2. The sort process closes the write end of the pipe and links the read end to become its standard input.

- Anything that ls writes to its standard output, sort would read from its standard input.

Limitations of pipes

1. Unix pipes are unidirectional channels from one process to another, and cannot be turned into bidirectional or multicast channels.

   This limitation makes pipes useless for complex inter-process communication.

2. The pipe system call creates a single “read end” and a single “write end”.

   Bidirectional communication can be simulated using a pair of pipes, but inconsistent buffering between both pipes can lead to a deadlock, especially if we only has control of the program on one end of the pipe.

Limitations of pipes

3. Pipes can only be shared between processes with a common ancestor.
4. This restriction prevents many useful forms of inter-process communication from being layered on top of pipes.
5. Pipes are a form of combination, there is no way for a process to name a pipe that it (or an ancestor) did not directly create via pipe.
6. Pipes cannot be used for clients to connect to long-running services.
7. Processes cannot open additional pipes to other processes that they already have a pipe to.

Files for Inter Process Communication

Two programs can communicate with one another by using an intermediate medium like a file.

```
davidkebo@linux:~$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos  code  src-cloud  src-cloud.zip
davidkebo@linux:~$ ls > content.txt
davidkebo@linux:~$ more content.txt
Desktop
Documents
Downloads
Music
Pictures
Public
Templates
Videos
code
content.txt
src-cloud
src-cloud.zip
```

The program `ls` sends its output to a file content.txt.
The program `more` takes content.txt as an input.
`ls` communicates with more via the file content.txt

FIFO (named PIPE)

FIFO is a first-in, first-out message-passing IPC in which bytes are sent and received as unstructured streams. It is also known as a named pipe.

The named pipe is a POSIX pipe in contrast to anonymous pipes created with the pipe() system call.

FIFOs work by attaching a filename to the pipe. Once created, any process (with correct access permissions) can access the FIFO by calling open() on the associated filename.

Once the processes have opened the file, they can use the standard read() and write() functions to communicate.

FIFO (named PIPE)

In Linux, we can create a FIFO with the commands mknod (using the letter “p” to indicate the FIFO type) or mkfifo.

C library functions – <sys/stat.h>

int mkfifo (const char *pathname, mode_t mode); Creates a new file identified by the pathname to use as a FIFO

```
davidkebo@linux:~/pipes$ mknod pipe2 p
davidkebo@linux:~/pipes$ mkfifo pipe1
davidkebo@linux:~/pipes$ ls -l
total 0
prw-rw-r-- 1 davidkebo davidkebo 0 Sep 11 17:33 pipe1
prw-rw-r-- 1 davidkebo davidkebo 0 Sep 11 17:33 pipe2
```

FIFO (named PIPE)

A common use for FIFOs is to create client/server applications on the same machine.

e.g.: An anti-virus server running in the background, scanning for infected files.

To get a report on potentially bad files, we run a client application that uses a FIFO to connect to the server.

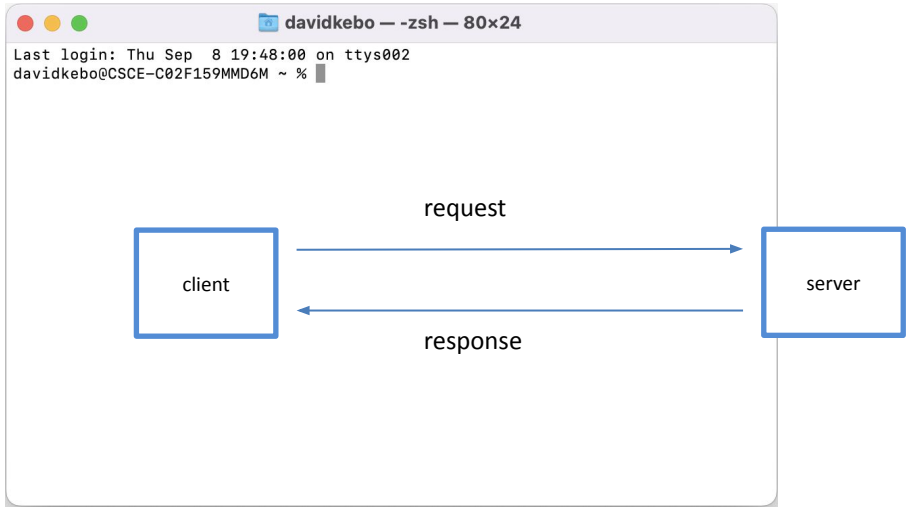
- The server and the client application are distinct processes running separate programs (not forked!).
- Instead, both processes use the name attached to the FIFO to set up the communication.

FIFO (named PIPE) example

Write as client-server program:

The server should print hello whenever the client writes a non-zero value to a file.

The server should shut down when the client writes a zero.



FIFO (named PIPE) example – Server

```c
int main(int argc, char const *argv[]) {

  /* Create the FIFO*/
  const char *FIFO = "/tmp/MY_FIFO";
  assert (mkfifo (FIFO, S_IRUSR | S_IWUSR) == 0);

  /* Open the FIFO. Delete FIFO if open() fails */
  int fifo = open (FIFO, O_RDONLY);
  if (fifo == -1)
  {
    fprintf (stderr, "Failed to open FIFO\n");
    unlink (FIFO);
    return 1;
  }

  /* Main server loop */
  while (1)
  {
    int req = 0;
    if (read (fifo, &req, sizeof (int)) != sizeof (int))
    continue;

    /* If we read a 0, quit; otherwise print hello */
    if (req == 0)
    break;
    printf ("hello\n");
  }

  /* Read a 0 from the FIFO, so close and delete the FIFO */
  close (fifo);
  printf ("Deleting FIFO\n");
  unlink (FIFO);

  return 0;
}
```

```
davidkebo@CSCE-C02F159MMD6M fifo % ./server
```

This code implements the server.

- The server prints hello when a client writes a non-zero value to a file and shuts down when the client writes a zero.
- The server starts by creating the FIFO with read and write permissions for the current user.
- The server, opens the FIFO in read-only mode and enters the listening loop.
- Once the server reads a value of 0 from the FIFO, it exits the loop, then closes and deletes the FIFO.

FIFO (named PIPE) example – Client

```
13  int main(int argc, char const *argv[]) {
14
15  const char *FIFO = "/tmp/MY_FIFO";
16
17  /* Use the file name to open the FIFO for writing */
18  int fifo = open (FIFO, O_WRONLY);
19  assert (fifo != -1);
20
21  /* Open the FIFO 6 times, writing an int each time */
22  for (int index = 5; index >= 0; index--)
23    {
24      /* Write 5, 4, 3, 2, 1, 0 into the FIFO */
25      int msg = index;
26      write (fifo, &msg, sizeof (int));
27
28      /* Add a slight delay each time */
29      sleep (1);
30    }
31
32  /* Close the FIFO */
33  close (fifo);
34
35    return 0;
36  }
```

```
davidkebo@CSCE-C02F159MMD6M fifo % ./client
```

```
davidkebo@CSCE-C02F159MMD6M fifo % ./server
hello
hello
hello
hello
hello
Deleting FIFO
```

This code implements the client.

- This client opens the FIFO, then writes a sequence of integers (5 down to 0) into the FIFO.
- Note that anything the client writes after the 0 would be thrown away, as the server would delete the FIFO at that point.

Limitations of FIFO

- Although FIFOs use standard file I/O functions (e.g., open(), read(), and write()), they are not regular files!
- Once data is read from a FIFO, the data is discarded and cannot be read again.
- With a regular file, multiple processes can read the same data from the same file. Regular files store data persistently, but FIFOs do not.
- FIFOs cannot be used for broadcasting a single message to multiple recipients; only one process can read the data.
- FIFOs (like pipes) are not suitable for bi-directional communication; if a process writes into the FIFO then immediately tries to read a response, it may read its own message.